IMPLEMENTING A NEW SYSCALL ON ARM'S ARCHITECTURE

Fabio Gritti , Sebastiano Mariani , Alessandro Loregiola

July 23, 2013

Contents

1 Introduction

1.1 Note

This is a project for the course "Piattaforme Software Per La Rete" of the university Politecnico Di Milano, under the supervision of the professor Alessandro Barenghi; The purpose of this project is to learning about the implementation of a new syscall on GNU/LINUX over an architecture based on ARM. All the documentation comes from various topic find on internet, from different book and from a bit of our previous acknoledge. We've make the decision to write this document in English because we think is important to internationalize the guide and maybe somebody can find useful our work (we sorry in advice for some grammar or sintax errors ...).

1.2 Details

We're going to start with a bit of general information about GNU/LINUX, this is our choice to permit at all the people to understand better all the related information that we'll explain later, then we'll proceed to illustrate the mechanism of the syscall and finally we'll explain how to implement a new syscall on x86 (the basic where all can take their tests) and then in ARM; All the code and the example referred to a 3.10 kernel for x86 and the "linux-rpi-3.6.y" downloaded from the official raspberry repository.

1.3 License

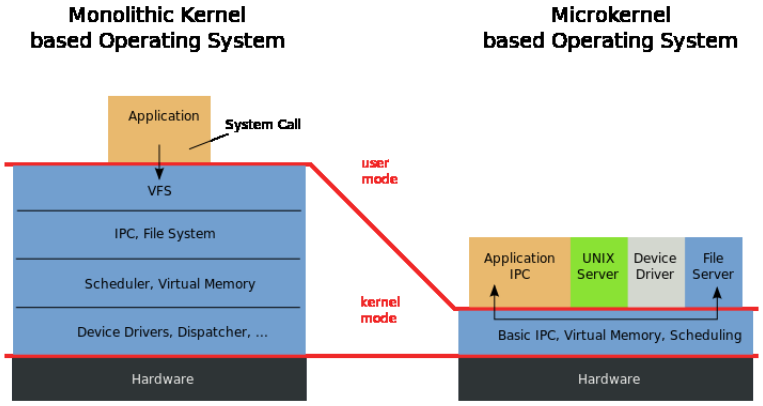
2 Begin

2.1 A little treasures

In this section we're going to present some basic point to understand the code that we'll show later. If you know all this information yet, you can skip easly this section.

2.1.1 Monolithic Kernel vs Microkernel

We can divide the kernel's design in two groups : the monolithic and the microkernel. Monolithic kernels are implemented as a single process running in a single address space; in this type of architecture the kernel can invoke functions directly beacuse all is running in kernel-space, the bad notice of this model is that if a module failed during its operation all the kernel is shut down. Microkernel are implemented with separated processes (servers) that are splitted into different address space and only the necessary servers are running in the privileged mode (the other are running in user-mode) ; the communication between servers naturally isn't direct, but is implemented via the IPC (interprocess communication) , likewise the separation of the various servers prevents a failure in one server from bringing down another. *Linux is a monolithic kernel, however borrows much of the good from microkernels*: modular design, capability to dynamicaly load separate binaries (modules) , but all runs in kernel mode, with direct function invocation.



Because the IPC mechanism involves quite a bit more overhead than a trivial function call, however, and beacuse a context switch from kernel-space to user-space or vice versa is often involved, message passing includes a latency and throughput hit not seen on monolithic kernels with simple function invocation.Consequently all pratical microkernel-based OS now place all of the most of their servers on kernel-space to avoid the latency of the frequently context-switch and gain the possibility to call function directly.

2.1.2 About the meaning of the symbol

When a big project is made up by a lot of C source files it's useful to declare variables with three different levels of visibility, specifically in the linux kernel that means:

- static: only visible within their own source code
- external: visible to any other code built into the kernel
- exported: visible and available to any loadable new module

When modules are loaded, they are dynamically linked into the kernel. As with userspace, dynamically linked binaries can call only into external functions that are explicitly exported for use. In the kernel, this is handled via special directives EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL().Functions that are exported are available for use by modules. Functions that are not exported cannot be invoked by modules. Exported symbols, of course, must be non-static. The set of kernel symbols that are exported are known as the exported kernel interfaces or even (gasp) the kernel API. Let's explain the functionality of exported symbols. For example, if you want to export a function/variable declared in one module you have to:

- add EXPORT_SYMBOL(variable/function_name) in the module where the variable/function defined. Export_symbol should be declared outside the function
- By declaring the function/variable as extern in other modules you can use these exported symbols
- During the insertion of module in to kernel, these exported variables/-functions are linked
- Ensure that the module inserted before the other modules start using them

Analize the list of symbols of an object file is possible, using utilities like nm. It's very important because the properties of declared symbols can be shown. Such as static variables don't show up in the table at all(having been declared as "static"); this table contains also global variables that can be distinguished by exported and not exported. The last ones have an entry in the module string table and symbol table, meaning that when the module is loaded, that symbols will be made available to other loadable modules. At last, besides the standard export macros defined in the kernel header file linux/module.h, there are a couple more specialized ones:

- EXPORT_UNUSED_SYMBOL() and EXPORT_UNUSED_SYMBOL_GPL(),which are used to identify symbols that are currently being exported but which are destined for unexporting some day.
- EXPORT_SYMBOL_GPL_FUTURE(), which denotes a symbol that is currently available to all modules, but is planned for restriction to GPL-only modules in the future.

2.1.3 What is asmlinkage

The asmlinkage tag tells gcc that that the function should not expect to find any of its arguments in registers (a common optimization), but only on the CPU's stack. Many kernel functions use the fact, that system_call consumes its first argument, the system call number, and leaves other arguments (which were passed to it in registers) on the stack. All system calls are marked with the asmlinkage tag, so they all look to the stack for arguments. The reason of using asmlinkage is that anyhow the kernel needs to save all the registers onto stack (in order to restore the environment before returning to the userspace) when handling the system call requests from userspace, so after that the parameters are available on stack.

2.1.4 What is a module

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

2.1.5 Modules vs Programs

A program usually begins with a main() function, executes a bunch of instructions and terminates upon completion of those instructions. Kernel modules work a bit differently. A module always begin with either the init_module or the function you specify with module_init call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides.All modules end by calling either cleanup_module or the function you specify with the module_exit call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered.Every module must have an entry function and an exit function. Since there's more than one way to specify entry and exit functions.

2.1.6 What is RCU

RCU is a synchronization mechanism that is optimized for read-mostly situations. For better understanding what is an RCU and how it works we can make a simple example: jump in a situation where we have a structure (e.g a struct with 3 integer fields) that is read from a lot of processes and updated from few processes (in best case only one); Unless the RCU if a process wants to read

the structure, but this is blocked by an updater the reader must wait and this can influence the performance in a kernel; With an RCU the reader hasn't got to wait because the mechanis provide it the old structure (obviously if an updater is modifing it) instead of blocking it waiting for the new one. When an updater decide to modify the structure it performs three step: removal, waiting and reclaim. In the first step it removes the pointer to the old version of the structure, then it waits for all the reader were registred before the synchronization and when the old readers have finished it removes phisically the old structure with a kfree. All the mechanism sounds like a "system versioning" inside your kernel specifically on a structure. During our project we found a brilliant guide to understand better the RCU system, if you are interested about this argoument you should absolutely read it: Guide to RCU.

2.2 First pratical instrument

2.2.1 How to load a module

In this section we'll only explain how to load a new module , *not* how to write it (for better understandig how to write a good module we remaind to the resources linked in the bibliography). Load a new module on linux kernel is very simple. First of all we have to create a small Makefile in the target folder and write the follow:

```
obj-m += <your_module_name_here>.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Then run *make* on your command line inside your target folder and you receive your <your_module_name_here>.ko file. With command *lsmod* you can show on your terminal all the modules that are running on your system and how many processes are using that, so let's add our module to the kernel using *insmod* :

```
insmod <your_module_name_here>.ko
```

Now if you launch *lsmod* again you can see your module at the top of the list! (probably alone, nobody is still using it, how sadly) When you want to remove it simply type :

```
rmmod <nome modulo>.ko
```

2.2.2 How to compile a kernel image

Here we are going to explain how to basic compile your new kernel image , there are many option and different modality , we'll only cover the basic. First of all you have to retrieve your copy of the kernel image from www.kernel.org, then extract the file from the archive downloaded and you receive the folder contains all the source. Go into the linux folder with your terminal and type *make menuconfig*[1], here you can custom your new kernel by adding or removing modules , when you are pleased of your configuration click on save. Finally the compiling phase: always in your linux folder type in terminal :

```
make -j4 > /dev/null
```

where 4 is the double of your number of core (this is for exploit the multiple jobs and save bit of time) , and the redirection in /dev/null is used to avoid the very verbose behavoir of the kernel's building (this phase can dure a lot of time depending on your pc). At the end of all the process you need to install the new kernel. How it is installed is architecture-and boot loader-dependent, consult the directios for your bootloader on where to copy the kernel image and how to set it up to boot. For example on an x86 architecture with GRUB , you would copy *arch/i386/boot/bzImage* to your /boot directory and naming it something like "vmlinuz-mylinuxkernel". Then you type on your terminal *sudo update-grub* and a new entry in the boot loader is automatically created. Installing modules is automated and architecture-independent. As root simply run: *make modules_install* , this install all the compiled modules to their correct home under */lib/modules*. Now reboot and you can see your new kernel in the grub menu.

2.2.3 How to cross-compiling

Despite cross-compiling seems to be an hard work to accomplish it is easier to do than to describe, we'll try to show you simple steps to do this on a x86 running GNU/LINUX (the steps are strongly dependent from OS / Architecture, in details we were running an ubuntu GNU/LINUX on x86 machine).

1. Open a terminal in your host machine and type

[1] *In case you miss the library simply type : apt-get install libncurses-dev*

```
make ARCH=arm CROSS_COMPILE=${CCPREFIX} menuconfig
```

```
mkdir workdir
cd workdir
```

this is the folder where we are going to download all our files: the rasp kernel and the tools for cross-compile.

2. Download the raspberry kernel (remember we are using the raspPi as our ARM architecture running 2013-05-25-wheezy-raspbian) Download rpi-3.6.y
3. Then download your copy of toolchain (these tools provides a modified version of gcc that can perform the cross-compilation ; the basic version of gcc compiles only on the architecture that you are working on)

```
git clone git://github.com/raspberrypi/tools.git
```

4. *Move into your linux's sources directory* from the terminal. Let us get the .config right.

```
cp arch/arm/configs/bcmrpi_defconfig .config
```

The above copy command will create a .config file with default configurations, if you want to make changes to the configurations then do a . CCPREFIX is an environment variable that contains the path to your compiler, for example we used *linaro* and our PATH was:

```
/home/<pcuser>/Desktop/workdir/tools
/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian
/bin/arm-linux-gnueabihf-
```

where pcuser is obviously your username and the file arm-linux-gnueabihf- is the arm-linux-gnueabihf-gcc: BE CAREFUL unless gcc and with '-'! (the type of cross-compiler (CC) that you use may be dependent of what your raspberry is running now, for example we were using 2013-05-25-wheezy-raspbian, so the right CC is *linaro)*

5. So our next step is to build the image : always in your linux's sources directory give the following command.

```
make ARCH=arm CROSS_COMPILE=${CCPREFIX}
-j< number of threads you want >
```

(the time of cross-compiling is strictly dependent by your pc)

6. So if everything goes fine, you should be able to get a kernel image in arch/arm/boot/Image. Image is the kernel image which we will be flashing via a ssh connection into the /boot folder by renaming it to kernel.img, but now keep calm and follow next point.

7. It's time to compile the modules. Let's create a new temp folder (like a container for the new modules) and export a new env. var OUTDIR (that point to that folder) . For example our is

```
/home/<pcuser>/Desktop/workdir/modules
```

8. Installing the modules in the temp directory (pointed by OUTDIR) by typing this command *into the linuxrasp folder*

```
make ARCH=arm CROSS_COMPILE=${CCPREFIX}
INSTALL_MOD_PATH="${OUTDIR}" modules_install
```

9. So now we need to move to the arch/arm/boot directory, and from there give the following command.

```
scp Image pi@<your raspberry ip local address>:
/home/pi/kernel.img
```

This would send your Image to /home/pi and renaming it 'kernel.img'. Now log into the raspberry pi and remove the 'kernel.img' existing in /boot directory. Finally copy the /home/pi/kernel.img in /boot.

```
cp Image /boot/kernel.img
```

Your new kernel is now in the right position and ready to running, but first...

10. You have to install the compiled modules (step 7) and the firmware. You can find both folder in your OUTDIR path in the existing dir. 'lib'. We suggest you to make two different .tar.gz: one for the folder 'modules' and another for 'firmware'. Once the .tar.gz are completed upload the archive in the /lib directory of raspberry with following command:

```
scp <module tar name> pi@<your raspberry ip local address> :/lib
scp <firmware tar name> pi@<your raspberry ip local address> :/lib
```

When all the files are arrived log into the raspberry and untar both the archives, this overwrite the existing folders modules and firmware with our new compiled files. (if you have errors from untar the archive we suggest you to create another tar and try again)

11. Finally reboot your raspberry and the new kernel should boot up! :)

3 System calls

3.1 What is

System call (syscall from now on) is how a process that runs in user mode request a service from operating system that runs in kernel mode. We can consider the syscalls as a layer between user space and kernel space that provides three purpose:

- Provides an abstracted Hardware interface for user space
- Ensure system security and stability
- Allows "process management"(forking ecc...)

A user space application cannot access services directly from the kernel because the kernel routines reside in kernel space and an application doesn't have permission to access this memory. (Definitely there would be a chaos if user-application can directly do this, and of course OS became not reliable) For example if an application wants to read from a device it has to go through the system calls provided by the Linux kernel in order to access at the raw function.

3.2 How it works

Syscalls are typically acessed via function calls defined in C library, they can define zero or more argument, and they provide a long type return value that indicates succes or errors.In Linux, each syscalls have a number called syscall number, and this number is used, when a user space process executes a syscall, to identifies which syscall run.The figure below describes how a syscall is invoked:

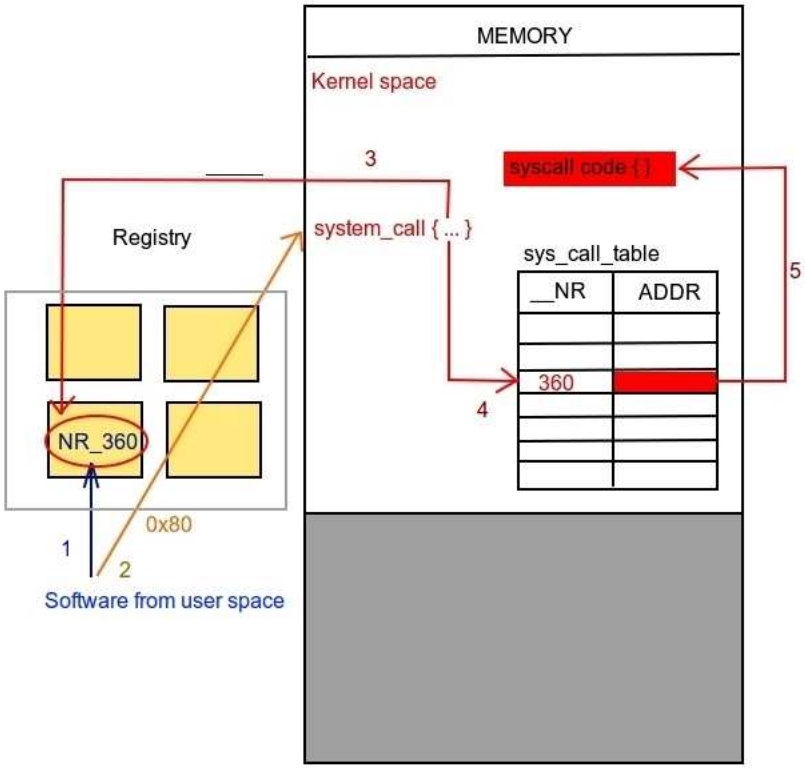


Figure 1: How system calls handler works

1. The user-process (for now on UP) puts the right syscall number in a register, for example on x86 it is passed via the eax register, instead on arm is passed via r7 register.

2. Then UP signals at the kernel that it wants to execute a syscall through a software interrupt (in x86 it is the x80 interrupt), this causes the system switching to kernel mode and executing the syscall handler.

3. The system_call() function checks the validity of the given syscall number in the register and eventually checks the input from user-space (if it exists).

4. Finally the syscall handler search in the sys_call_table the row corresponding at the NR found in the register and jump at the address found in this table.

5. The routine at the address is executed.

3.3 Example : a small 'wrapper' to sys_open (on x86)

For better undersanding the mechanism of the system call handler we use a simple kernel module called spy _module. This module change the behavoir of the syscall 'sys_open' and execute our piece of code ('our_sys_open') before executing the real 'sys_open', without the kernel sends any errors (despite you must be root for load the module you can image the power of this type of "syscall overwriting" and all related security issue) here's the code of the spy_modules:

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>
#include <linux/unistd.h>
#include <linux/syscalls.h> // necessary for the symbol sys_close
#include <linux/sched.h> // for struct task_struct
#include <asm/uaccess.h>
#include <asm/thread_info.h>
#include <linux/cred.h>  //necessary for achieve the structure that
     store the credential pf the process

/* my pointer to the table sys_call_table */
unsigned long **sys_call_table;

/* a pointer to a function. I'm going to use it as a store of the real
     address of the OPEN */
asmlinkage long (*original_call) (const char *,int,int);

/* Necessary for receive parameters from command line */
static int uid; module_param(uid, int, 0644); //<-- the uid, type,
     permission

MODULE_PARM_DESC(uid , "User id to spy");
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Fabio Gritti, Sebastiano Mariani, Alessandro Loregiola");

/* Function used to aquire the address of the sys_call_table since it is
      not long exported as a symbol */
static unsigned long **aquire_sys_call_table(void) {
        unsigned long int offset = PAGE_OFFSET;
        unsigned long **sct;

        /*We're moving row by row in the sys_call_table*/
        while (offset < ULLONG_MAX) {
                sct = (unsigned long **)offset;

                if (sct[__NR_close] == (unsigned long *) sys_close)
                        return sct;

                /*each element in the table is 32 bit ( on x86-32 )
                  so we are increasing the address in our offset by 4
                     bytes
          ( in fact sizeof(void*) returns 4 )
        */
                offset += sizeof(void *);
        }

        printk(KERN_ALERT "Failed to load the sys_call_table");
        return NULL;
        }

/* our piece of code first the real calling of the open */
asmlinkage int our_sys_open(const char *filename , int flags , int mode
     ) {
        int i = 0;
```

```
        char ch;

         /*
           Check if this is the user we're spying on
        the attribute uid has been moved to a structure cred in linux/
            cred.h, so I
          take this from current(the current task_struct)->cred(pointer
              to the cred structure)->uid
         */
         if (uid == current->cred->uid) {

                /*
        Oh, a process of our user!
                Report the file!
                */
                 printk("Opened file by %d: ", uid);

                 do {
                         get_user(ch, filename + i);
                         i++;
                         printk("%c", ch); // printing the file path
                         }
                 while (ch != 0);

                 printk("\n");  }

                 /*
                 Then jump to the real address of the open function! (
                      stored in the init of this module )
                 */
                 return original_call(filename,flags,mode);
         }


/*
enable and disable page protection of kernel's pages
*/
static void enable_page_protection(void)  {
         unsigned long value;
         asm volatile("mov %%cr0, %0" : "=r" (value));

         if((value & 0x00010000))
                 return;

         asm volatile("mov %0, %%cr0" : : "r" (value | 0x00010000));
         }

static void disable_page_protection(void)  {
         unsigned long value;
         asm volatile("mov %%cr0, %0" : "=r" (value));

         if(!(value & 0x00010000))
                 return;

         asm volatile("mov %0, %%cr0" : : "r" (value & ~0x00010000));
         }

static int __init startup_init(void) {
         printk (KERN_ALERT "spy_module inserted correctly!");

         if(!(sys_call_table = aquire_sys_call_table()))
            return -1;

         disable_page_protection();

         original_call = (void *)sys_call_table[__NR_open];

    sys_call_table[__NR_open] = (unsigned long *)our_sys_open;
```

```
        enable_page_protection();

    return 0;
        }

static void __exit shutdown_exit(void) {
        if(!sys_call_table)
                return;
        disable_page_protection();

        sys_call_table[__NR_open] = (unsigned long *)original_call;

        enable_page_protection(); printk("Module closed");
        }

/* classic */
module_init(startup_init);
module_exit(shutdown_exit);
```

To do this we have "tricked" the syscall handler in this manner

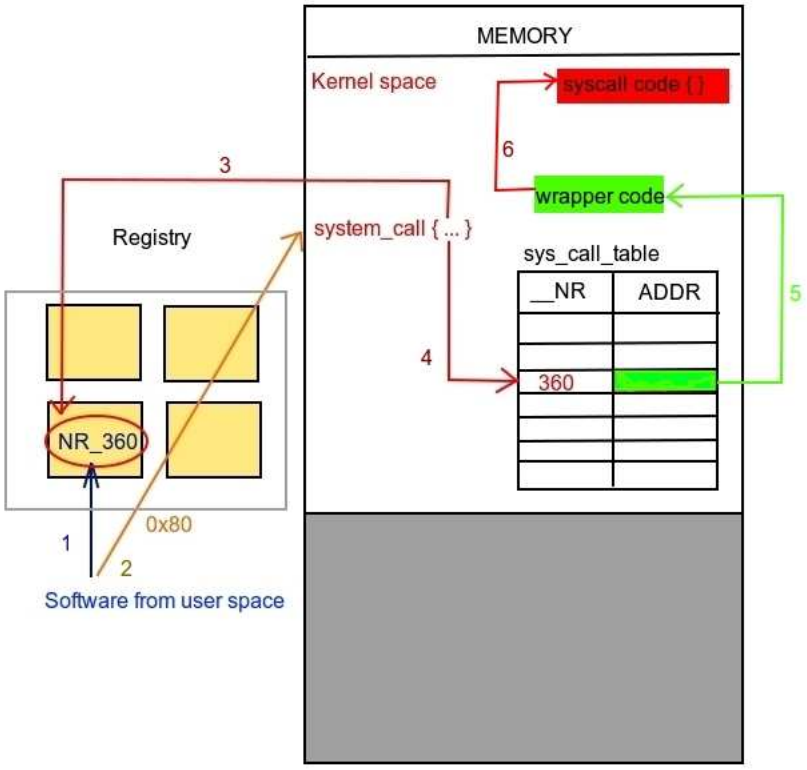


Figure 2: Trick of handler

1. A User-space process puts the number of sys_open in the register
2. The software interrupt calls the syscall handler.

3. The system_call() function check the number.
4. Retrieve the corresponding address of the __NR called from sys_call_table
5. Execute the function at the address found , that is not 'sys_open' but 'our_sys_open'
6. The return value of 'our_sys_open' is a function pointer that makes the system jump to the address of the original syscall 'sys_open' stored before.

The user will not notice additional operation before the correct execution of sys_open. Let's explain the code: when module is insmoded we aquire the syscalltable and perform these operations:

1. Disable page protection
2. Store the real address of the open in the original_call
3. Change the row at __NR_open with our_sys_open
4. Re-enabling page protection!

Enable and disable kernel's page protection sounds bit aggressive, but it is necessary because the memory where kernel is living is a read only memory, this implies we can't write anything in the syscalltable. The two function, *static void enable_page_protection(void)* and *static void disable_page_protection(void)*,provide a trick to bypass this limitation, disabling via hardware the control of the read-only pages by changing the 16 bit of processor that control this behavoir (in x86). When module is rmmoded we roolback the situation of the sys_call_table to normality by re-changing the row __NR_open to the original address. Usage of this module:

1. Compile the module with the right Makefile.
2. Type in shell (with root privileges): insmod < module name >.ko uid = < number of id to spy >
3. Watch the spying printk with the dmesg command
4. Type in shell (with root privileges) rmmod <module name >.ko

3.4 How to add a new syscall on x86

3.4.1 Create the new syscall

These are the steps for add a new system call in your linux kernel on x86 architecture (32 bit); This procedures working on a 3.10 kernel, previous version has got bit different step that you can find easily online.

1. You have to edit the *linux-3.10/arch/x86/syscalls/syscall_32.tbl* file with a new row following the existing syntax : < number > <abi> <name> < entry point >, for example you have to write:

```
360   i386    newcall     sys_new_call
```

where 360 is obviously the next number after the last syscall and the __NR that you're going to use when you want to call the new syscall.

2. Edit *linux-3.10/include/linux/syscalls.h* file with a new row that will be the prototype of your syscall.

```
asmlinkage long sys_new_call ( int num )
```

where num is the parameter of our syscall (if you have a parameter).

3. Write a new C file in *linux-3.10/arch/x86/kernel* where you put the real code of the system call (what it is going to do!) eg:

```
#include <linux/linkage.h>
#include <linux/printk.h>
SYSCALL_DEFINE1(new_call,int , i) {
printk("<0> My syscall called");
return i+2;
}
```

SYSCALL_DEFINE is a macro used for define a new systemcall, if you have one parameter to pass write SYSCALL_DEFINE1, two parameter SYSCALL_DEFINE2 etc... The first value of the macro is the syscall's name , then if you have parameters you have to write the type and the name of each ones .

4. Edit the Makefile in *linux-3.10/arch/x86/kernel* by adding a new line

```
obj-y += new_call.o
```

5. Recompile the kernel! (you have to make this operation because the sys_call_table is automatically generated by a script during the kernel's compilation, you can't add dinamically a new row to this table)

3.4.2 Use the new syscall

To test the new syscall just created you have to write a small C program that call it, the following example show you how to do it.

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#define __NR_sys_new_call 360

long newcall(void) {
return syscall(__NR_sys_new_call , 1 );
}

int main(int argc, char *argv[]) {
int retval=4;
int ret;
printf("retval at start is %d\n" , retval);
ret = newcall();
printf("From my call I receive : %d \n ", ret);
retval+=ret;
printf("ret + retval is %d\n", retval);
if(retval < 0)   {
perror("My system call returned with an error code.");
}
}
```

Where syscall function is used to put the __NR into register and jump to the syscall handler and the '1' is the parameter of our syscall.

3.5 How to add a new syscall on ARM

3.5.1 Create the new syscall

These are the steps for add a new system call in your linux kernel on ARM architecture (32 bit); This procedures is tested against linux-rpi-3.6.y, the new version linux-rpi-3.8.y has changed some things, but the general steps should remain these:

1. Edit the file *linux-rpi-3.6.y/arch/arm/include/asm/unistd.h* with a new row under the other syscall like this

```
[ ... ]
_NR_process_vm_writev (__NR_SYSCALL_BASE+377)
                      /* 378 for kcmp */
_NR_newcall           (__NR_SYSCALL_BASE+379)
```

   Where 379 was our next number from the last syscall.

2. Edit the file *linux-rpi-3.6.y/include/linux/syscalls.h* with a new line at the end, this is the prototype of our function.

```
asmlinkage long sys_newcall(<type par> <name par> ... )
```

3. Edit the file *linux-rpi-3.6.y/arch/arm/kernel/calls.S* with a new row under the last CALL

```
[ ... ]
CALL(sys_process_vm_writev)
CALL(sys_ni_syscall) /* reserved for sys_kcmp */
CALL(sys_newcall)
```

4. Copy the source of your syscall in *linux-rpi-3.6.y/arch/arm/kernel*

5. Edit the Makefile *linux-rpi-3.6.y/arch/arm/kernel/Makefile* with a new line

```
obj-y +=newcall.o
```

3.5.2 Our project example: syscall that close all files of a process

Here is the code (followed by explanation) of our project: writing a syscall that in a safe manner closes all the open files of a process. In this version of the program the stdin,stdout,stderror (0,1,2) are closed too (If you want close only the file with id number >=3 initialize j=3)

```
#include <asm/uaccess.h>
#include <linux/cred.h>
#include <linux/linkage.h>
#include <linux/printk.h>
#include <linux/sched.h>
#include <linux/syscalls.h>
#include <linux/export.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/mmzone.h>
#include <linux/time.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <linux/file.h>
#include <linux/fdtable.h>
#include <linux/bitops.h>
#include <linux/interrupt.h>
#include <linux/spinlock.h>
#include <linux/rcupdate.h>
#include <linux/workqueue.h>


static struct task_struct* get_task_by_pid(pid_t pid)
{
return pid_task(find_pid_ns(pid, task_active_pid_ns(current)), PIDTYPE_PID);
}

static void close_files(struct files_struct * files)  {
int i, j;
struct fdtable *fdt;
j = 0;
rcu_read_lock();
fdt = files_fdtable(files);
rcu_read_unlock();
```

```
for (;;)
{
i = j * BITS_PER_LONG;
if (i >= fdt->max_fds)
   break;
set = fdt->open_fds[j++];
while (set) {
          if (set & 1) {
          struct file * file = xchg(&fdt->fd[i], NULL);
          if(file) {
                    filp_close(file, files);
                    cond_resched(); }
          }
         i++;
          set >>= 1;
}}}

SYSCALL_DEFINE1(defclose , pid_t , pid)
{
        struct task_struct *result = NULL;
        if( pid <=1 )
           return -1;
        rcu_read_lock();
        result = get_task_by_pid(pid);
        rcu_read_unlock();
        close_files(result->files);
}
```

First of all: there are a lots of library (maybe some are useless, but the carefulness is never enough in kernel space when a syntax error means "recompile again" that cause a waste of time). Let's explain the mechanism of our code:

- We control the validity of the pid, negative pid are refused and the syscall return with an error. (init process is protected too from closing its files)
- If all is right , then we call an rcu_read_lock and we use the exported function "get_task_by_pid" for retrieving the process descriptor of the process identified by the pid (for more information see its implementation in the kernel source)
- When "get_task_by_pid" returns our process descriptor, an rcu_read_unlock is called and then we call the function close_files that takes the argument *result->files*. What is *result->files*? Result is obviously the process descriptor returned from the previous function and files is the pointer to the files_struct of the process (that store all the information of the open file).

- At this point closing files is demanded to this function *close_files*. The first step is to receive the array that contains all the open files: the *struct fdtable* (with the usually caution of the RCU). Then the istruction

```
i = j * BITS_PER_LONG
```

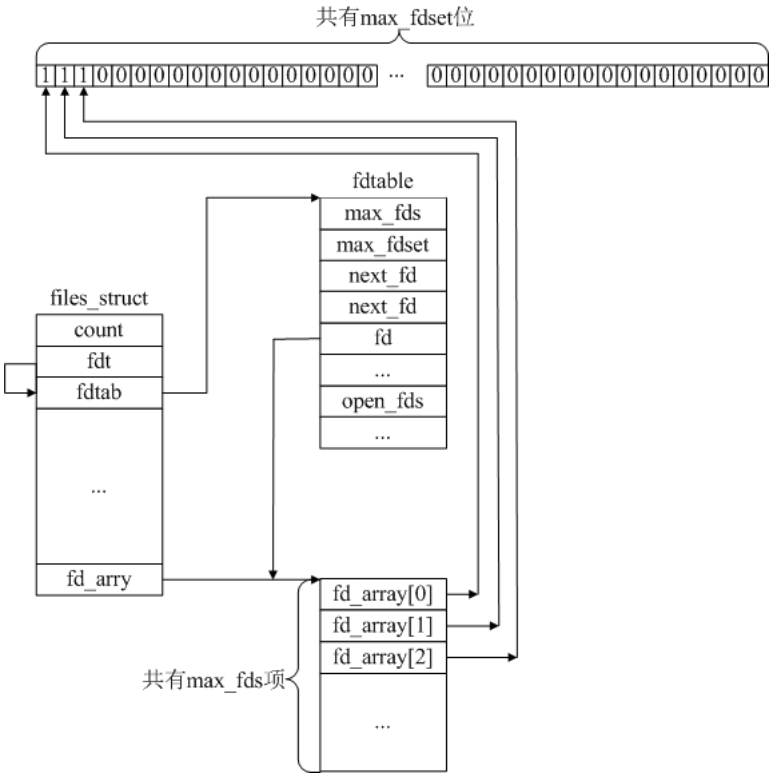
is used for correctly move our pointer through the table aquired. The control

```
if (i >= fdt->max_fds) break;
```

is used for understand if we've reached the end of the table (and with break we exit this function), next to this the line

```
set = fdt->open_fds[j++];
```

is used to receive the open_fds: the bitmap of all the open files (*Open_fds* is a pointer to a bit field that manages the descriptors of all currently opened files.There is just one bit for each possible file descriptor;if it is set to 1, the descriptor is in use;otherwise it is unused) . Then if we find an used descriptor (set & 1 !=0 means that the fd is in use) we call the xchg; that function puts in &fdt->fd[i] the value NULL (the file is now logical closed beacuse its descriptor is set to null) , and then return the *struct file* of the descriptor at &fdt->fd[i]; this descriptor is then passed to the filp_close that close concretely the file (check the kernel documentation for this function). For better understanding the data structures associated with a process see this picture:



4 Notes

We've tried various time to put our modded kernel on a raspberry running archlinx, but we encountered various problem during the module loading. Unless the modules the kernel can run our syscall unless problem, but when we tried to load our compiled modules and firmware we obtained a kernel error: oops 17 and a NULL deference of a memory pointer. Maybe a kernel bug?

References

[1] http://stackoverflow.com

[2] http://elinux.org/RPi_Kernel_Compilation

[3] Robert Love, Linux Kernel Development (Third Edition)

[4] http://mentorlinux.wordpress.com

[5] http://www.tldp.org

[6] http://lxr.linux.no

[7] http://www.linux.com

[8] http://www.rdrop.com

[9] http://www.advancedlinuxprogramming.com